과제 #3 : 동적 메모리 관리

○ 과제 목표

- 프로그램에서 메모리를 동적으로 할당하는 사용자 지정 메모리 관리자를 구현
- malloc(3) 및 free(3)와 같은 C 라이브러리 함수를 대체하는 역할을 하는 메모리 할당하고 해제하는 함수 구현을 통해 메모리 관리 기법 이해

○ 기본 지식

- Linux의 프로세스 관련 시스템 호출 함수 (mmap(2), munmap(2) 등)

○ 과제 기본 내용

- mmap(2)을 사용하여 OS가 메모리 페이지를 가져오고, 메모리를 요청 시 메모리 페이지들의 chunk를 동적으로 할당
- 사용자 입력을 위해 잠시 중지하거나 sleep() 등으로 오랫 시간 동안 실행되는 간단한 C 프로그램을 작성하고 실행
  - ✓ 실행된 프로세스가 활성 상태(메모리 내에서 runnable 상태)인 동안 ps(1) 또는 다른 유사한 명령어를 사용하여 해당 프로세스의 메모리 사용량 측정
    - 프로세스의 가상 메모리 크기 (VSZ)와 프로세스의 상주 세트 크기 (RSS) (프로세스에 할당 된 물리적 RAM 페이지 만 포함) 측정
    - /proc 파일 시스템에서 적절한 파일에 액세스하여 Linux /proc 파일 시스템 내 프로세스의 다양한 메모리 이미지를 보여줌
- 작성한 간단한 C 프로그램에 OS의 빈 페이지를 메모리 맵에 추가하는 코드를 수정 구현하고 수정한 간단한 C 프로그램을 일시 중지하여 해당 프로세스에서 사용하는 가상 및 실제 메모리를 측정
  - ✓ OS 커널에게 private 모드(페이지를 다른 프로세스와 공유하지 않는)의 anonymous 페이지(디스크의 파일을 지원하지 않는 메모리, malloc()에 의해 할당된 메모리. 내부적으로 malloc(3)은 sbrk(2), brk(2) 또는 mmapp(2)을 호출하여 커널에 anonymous 페이지 요청) 요청
  - ✓ 메모리 매핑된 페이지로는 다른 작업을 수행하면 안됨.
- 마지막으로 일부 데이터를 memory mapped 페이지에 쓰고 가상 및 실제 메모리 사용량을 다시 측정

○ 과제 구현 내용

- 1. 간단한 메모리 관리자(동적으로 메모리 할당 및 해제) 구현
  - ✓ alloc.c를 직접 구현
  - ✓ 주어진 alloc.h는 구현해야 하는 함수를 선언함. 수정할 필요 없음.
  - ✓ 메모리를 동적으로 할당 및 해제 (아래 두 방법 중 하나 써도 되고, 혼합해서 써도 됨)
    - 방법 1: 배열 이용. mmap(2)으로 4KB 전체를 잡고, 요구한 만큼 주소값 더함. 여기서 할당된 주소와 할당 여부를 구조체 배열로 관리
    - 방법 2: 링크드 리스트 이용. mmap(2)으로 4KB 전체를 잡고, 할당된 메모리 주소 영역을 링크드 리스트로 관리. 별도로 할당되지 않은 메모리 주소 영역을 링크드 리스트로 관리
  - ✓ 메모리 관리자는 OS에서 mmap(2)을 통해 4KB 페이지를 요청하고 4KB의 메모리를 관리. 8 바이

트의 배수 인 크기로 할당 및 해제

- mmap(2)을 사용하여 OS 커널이 메모리 페이지를 가져오고, 메모리를 요청 시 메모리 페이지들의 chunk를 동적으로 할당

✓ 구현해야 하는 기본 함수 설명

- (1) init_alloc() : mmap(2)을 통해 OS에서 4KB 페이지를 할당하고 필요한 다른 데이터 구조를 초기화하는 등 메모리 관리자 초기화. 이 함수는 메모리 관리자에게 메모리를 요청하기 전에 사용자가 호출. 성공 시 0을 리턴. 에러 시 0이 아닌 오류 코드 리턴
- (2) cleanup() : 메모리 관리자의 상태를 정리하고 memory mapped 페이지 OS 커널에게 다시 반환. 성공 시 0을 리턴. 에러 시 0이 아닌 오류 코드 리턴
- (3) alloc(int) : 할당할 integer buffer 크기를 갖고 성공 시 버퍼에 대한 포인터를 리턴. 에러 시 NULL 리턴
  - ☞ 에러의 예 : 요청 된 크기가 8 바이트의 배수가 아니거나 여유 공간이 부족한 경우. 성공시 반환 된 포인터는 메모리 관리자의 4KB 페이지 내에서 유효한 메모리 주소를 가리켜야 함
- (4) dealloc(char *) : 이전에 할당 된 메모리 chunk에 대한 포인터로 전체 chunk를 해제

✓ alloc()를 구현하기 위해 malloc(3)과 같은 라이브러리 함수를 사용해서는 하면 안됨. 대신 mmap(2)을 통해 OS에서 페이지를 가져 와서 malloc(3)과 같은 기능을 직접 구현

✓ 메모리 관리자는 다양한 방법으로 구현 가능. 따라서 다음 5가지 기능이 제공될 수 있게 자유롭게 설계하고 구현해도 됨.

✓ 5가지 기능

- (1) 메모리 관리자는 init_alloc()를 통해 사용자에게 할당 할 수 있는 전체 4KB 생성.
  - ☞ 사용 가능한 메모리 양을 줄일 수 있도록 헤더 또는 메타 데이터 정보를 페이지 자체에 저장해서는 안됨.
  - ☞ 할당 크기를 추적하는 데 필요한 모든 메타 데이터는 코드에 정의 된 데이터 구조 내에 있어야 하며 메모리 매핑 된 4KB 페이지 자체에 포함되지 않아야 함
- (2) 이중 할당 금지. 이미 할당 된 메모리 영역은 사용자가 해제 할 때까지 다른 할당에 사용할 수 없음.
- (3) 크기가 N1 바이트 인 메모리 chunk가 해제되면 향후 N2 크기의 메모리 할당에 사용할 수 있어 야 함. 단, N2 ≤ N1.
  - ☞ N2 <N1이면 크기 N1-N2의 남은 chunk를 향후 할당에 사용할 수 있어야 함. 즉, 메모리 관리자는 할당을 위해 더 큰 chunk를 더 작은 chunk로 분할 할 수 있어야 함
- (4) 크기가 N1 및 N2 인 두 개의 free 메모리 chunk가 서로 인접 해있는 경우, 크기 N1 + N2의 병합 된 메모리 chunk를 향후 할당에 사용할 수 있어야 함. 인접한 메모리 chunk를 병합하고 더 큰 chunk를 할당할 수 있어야 함
- (5) 여러 번 할당하고 해제한 후에 4KB 페이지에 할당된 chunk와 사용 가능한 chunk가 메모리 상에 흩어져 있을 수 있음. chunk 할당 요청이 오면 빈 chunk를 잘 리턴할 수 있는 메모리 할당 방식 (예 : 최적 적합, 최초 적합, 최악 적합 등)을 선택하여 할당

✓ 구현 테스트

- 제공한 test_alloc.c 이용
  - ☞ test_alloc.c는 메모리 관리자를 초기화하고 사용자가 구현 한 alloc() 및 dealloc()를 호출하는 여러 테스트를 실행
  - ☞ $ gcc test_alloc.c alloc.c
  - ☞ $ ./ a.out
- 구현한 코드를 테스트하기 위해 더 많은 테스트 프로그램을 스스로 작성해서 테스트해도 됨

✓ 메모리 관리자에 필요한 기능이나 자료구조는 테스트 프로그램에 포함시키면 안됨. 전체 메모리 관리 코드는 alloc.c에만 포함되어야 함.

- 2. 확장 가능한 Heap

✓ Memory mapped 페이지에 사용자 지정 메모리 할당자 구현
✓ 프로세스가 요청 시에만 Memory mapped 페이지 할당
✓ elloc.c를 직접 구현
✓ 주어진 ealloc.h는 구현해야 하는 함수를 선언함. 수정할 필요 없음.
✓ 4가지 기능
- (1) init_alloc() : 메모리 관리자를 초기화. 필요한 데이터 구조를 초기화할 수 있음. 그러나 요청 시에만 메모리를 할당해야하기 때문에 여기서는 페이지를 메모리 매핑하면 안됨
- (2) cleanup() : 메모리 관리자의 모든 상태 정리. 메모리 할당이 이루어질 때 mmap()를 호출하지만 munmap()을 통해 OS 커널로 메모리를 반환하지 않는다고 가정하기 때문에 여기서 OS 커널로부터 어떠한 Memory mapped 페이지를 해제할 필요 없음.
- (3) alloc(int) : 할당할 integer buffer 크기를 갖고 성공 시 버퍼에 대한 포인터를 char * pointer로 리턴. 에러 시 NULL 리턴.
  ☞ 요청 된 버퍼 크기는 256 바이트의 배수이며 4KB (페이지 크기)를 넘지 않아야 함. 총 할당 된 메모리는 4 페이지, 즉 16KB를 초과하지 않음.
  ☞ 할당 된 모든 chunk가 4 페이지 중 하나에 완전히 상주한다고 가정하기 때문에 페이지 경계를 넘어서 chunk를 할당하는 것에 대해 신경 쓰지 않아도 됨.
  ☞ 할당 요청을 받으면 메모리 할당자는 기존 페이지 중에서 사용 가능한 메모리 chunk가 있는지 확인. 그렇지 않은 경우 mmap(2)을 호출하여 OS 커널에서 anynomous private 페이지를 할당. 필요 시 OS 커널에서 4KB 페이지 단위로 메모리를 요청해야 함. 그러나 메모리 할당자는 OS 커널에서 필요한 것보다 더 많은 페이지를 메모리 매핑해서는 안되며, 할당 요청을 충족하는데 필요한 만큼의 페이지만 요청
  ☞ 예를 들어, 메모리 할당자가 초기화되고 메모리 할당 자의 사용자가 1024 바이트를 할당하기 위해 alloc(1024)를 호출했다고 가정하면, 메모리 할당자는 이 시점에서 첫 번째 mmap(2)를 호출하여 하나의 4KB 페이지만 Memory mapping 해야 함. 두 번째 페이지를 할당하기 위한 다음 mmap(2) 호출은 다음 할당 요청을 충족하기 위해 첫 번째 메Memory mapped 페이지 내에 free 공간이 없는 경우에만 가능. mmap(2)에 대한 연속 호출은 모든 시스템에서 가상 주소 공간의 연속적인 부분을 리턴하지 않을 수 있다는 점에 유의.
- (4) dealloc (char *) : 이전에 할당 된 메모리 chunk(이전 alloc() 호출에 의해 리턴된)에 대한 포인터를 가져와 해당 chunk 해제
  ☞ 할당 해제 시 힙을 축소 할 필요 없음. munmap(2)를 통해 해제된 빈 페이지를 OS 커널에게 반환할 필요가 없음.
- (5) 할당된 chunk와 사용 가능한 chunk의 병합은 1 과 동일

✓ 구현 테스트
- 주어진 test_ealloc.c 이용
  ☞ test_ealloc.c는 메모리 관리자를 초기화하고 사용자가 구현 한 alloc() 및 dealloc()를 호출하는 여러 테스트를 실행
  ☞ $ gcc test_ealloc.c ealloc.c
  ☞ $ ./ a.out
  ☞ 사용자 정의 메모리 할당자를 사용하여 다중 할당 및 할당 해제를 수행하고 할당된 메모리의

상태 확인.

- ☞ 테스트 스크립트는 할당 요청을 충족하기 위해 기존 사용 가능한 chunk를 올바르게 할당하고 병합하는 확인
- ☞ OS에서 요청 시에만 메모리 매핑 페이지인지 확인하기 위해 테스트 프로그램은 프로세스의 가상 메모리 크기 (VSZ)도 주기적으로 출력
- ☞ 단, 물리적 메모리 할당은 사용자가 제어 할 수 없으며, OS 커널의 요구 페이징 정책에 의해 완전히 처리되기 때문에 프로세스에서 사용하는 실제 물리적 메모리가 아닌 VSZ 값의 정확성만 확인 가능

○ 과제 제출물 및 마감

- 2020년 11월 22일 (일) 23시 59분 59초까지 제출
- 제출물
  - ✓ 개요 또는 구현 내용 (수정한 부분 간단하게 설명)과 실행 결과 캡쳐 등 자율적으로 보고서 작성
  - ✓ 제출할 tgz(또는 zip) 내에는 (1) 보고서 파일과 위 세부 과제 1, 2의 구현 결과물을 넣을 (2) alloc 디렉토리와 (3) ealloc 디렉토리가 있어야 함.
  - alloc 디렉토리에는 수정한 alloc.c와 alloc.h 존재해야 함. (실행파일 및 수정하지 않은 파일은 넣지 말 것). 단, 별도로 테스트 프로그램을 만들 경우는 .c 파일과 실행 파일을 넣기 바람.
  - ealloc 디렉토리에는 수정한 ealloc.c와 ealloc.h 존재해야 함. (실행파일 및 수정하지 않은 파일은 넣지 말 것). 별도의 테스트 프로그램 만들 필요 없음.
- 마감 : 2020년 11월 22일 (일) 23시 59분 59초까지 구글클래스룸으로 제출

○ 필수 구현

- 1

○ 배점 기준

- 1 : 50점
- 2 : 50점